Estruturas de Dados

Listas

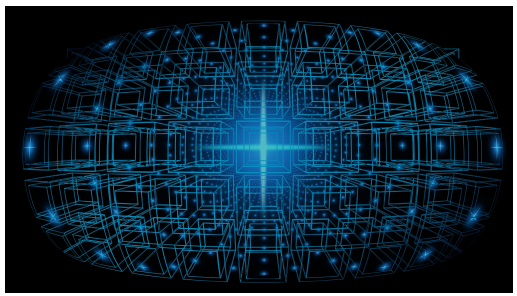
Luiz Henrique de Campos Merschmann
Departamento de Computação Aplicada
Universidade Federal de Lavras

luiz.hcm@ufla.br

Na Aula de Hoje

Estruturas de Dados

O que são estruturas de dados?



Uma estrutura de dados é uma forma particular de armazenar e organizar os dados que serão utilizados por um programa de modo que eles possam ser manipulados de maneira eficiente.

O que estudaremos?

Nesta disciplina estudaremos algumas das principais estruturas de dados[1] e veremos como utilizá-las em Java. São elas:

- Listas, Pilhas, Filas e Tabelas Hash.

[1] O objetivo aqui não é implementá-las, mas sim entender como funcionam e saber utilizá-las.

Estruturas de Dados em Java

- O pacote *java.util* fornece classes que implementam diversas estruturas de dados (chamadas em Java de **coleções**) que podemos usar em nossos programas.
- Mas o que é uma **coleção**?
  - Podemos dizer que é um conjunto de objetos de mesma natureza.
  - Exemplos: coleção de músicas, coleção de carros, coleção de moedas etc.
- Em Java, uma **coleção** é representada por uma **classe**. Portanto, as **operações sobre coleções** são realizadas a partir dos **métodos** implementados nessas classes.
- Quais operações podemos fazer em uma coleção?
  - **Adicionar** um novo objeto na coleção.
  - **Remover** um objeto da coleção.
  - **Organizar** os objetos da coleção segundo algum critério.

UFLA UNIVERSIDADE FEDERAL DE LAVRAS

Relembrando...

Antes de conhecer as classes que implementam as estruturas de dados, vamos relembrar de algo que já utilizamos...

O que nós já utilizamos para armazenar conjuntos de dados de mesmo tipo em nossos programas?

Nós já utilizamos os vetores (ou *arrays*) para armazenar conjuntos de dados de mesmo tipo!

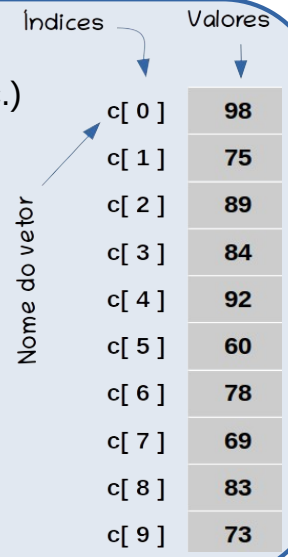
Então vamos relembrar algumas coisas importantes...

- Em Java, os vetores são considerados objetos.
- Um vetor pode conter elementos de tipos primitivos (*int*, *float*, *double* etc.) ou de tipos por referência (para objetos).
- Exemplo de declaração de um vetor: `int[] c = new int[10];`
- Exemplo de modificação dos valores de um vetor de inteiros:

```
for(int i = 0; i < c.length; i++){
    c[i] = i;
}
```

- Exemplo de vetor passado por parâmetro para método:

```
public void definirVetor(c[] int){
    for(int i = 0; i < c.length; i++){
        c[i] = i;
    }
}//Passagem por referência, ou seja, estaremos alterando o
  vetor passado por parâmetro.
```



Desvantagem dos Vetores

Apesar da sua grande utilidade, os vetores têm uma limitação: são estruturas de dados estáticas!

E o que isso significa?

Significa que o seu tamanho, definido no momento de sua declaração, é fixo.

Entendi… E como eu faço se eu não souber quantos elementos eu terei que adicionar em um determinado vetor?

Nesse caso, a única opção é “estimar” o tamanho necessário do vetor. Mas com isso, duas situações podem ocorrer:

1) **Desperdiço de memória** se você estimar um tamanho maior do que é realmente necessário.
2) **Não conseguir armazenar todos os dados** se você estimar um tamanho menor do que realmente é necessário.

Para resolver este problema podemos usar as **coleções** do Java!

Classe *ArrayList*

- A classe **ArrayList** implementa uma **lista**!
- Nessa lista, a ordem dos seus elementos é a mesma da inserção e o acesso aos mesmos pode ser feito pela sua posição.
- A classe de coleção **ArrayList<T>** fornece uma solução conveniente quando necessita-se de uma estrutura que possa **alterar dinamicamente seu tamanho** para acomodar mais elementos.
  - O **T**[2] corresponde ao tipo dos elementos que você deseja armazenar no **ArrayList**.

    Ex.: **ArrayList**<String> *nomes*;

    Nesse caso estamos declarando a variável *nomes*, do tipo ArrayList de *strings*, ou seja, para armazenar elementos do tipo *String*.

[2] Classes que dependem de um segundo tipo são chamadas de classes genéricas.

Classe *ArrayList*

Para instanciar um *ArrayList* podemos fazer da seguinte maneira:

```
ArrayList<String> nomes = new ArrayList<String>();
```

OU

```
ArrayList<String> nomes;
nomes = new ArrayList<String>();
```

Seus principais métodos são:

add(x)	Adiciona o elemento *x* ao final do *ArrayList*.
clear( )	Remove todos os elementos do *ArrayList*.
contains(x)	Retorna true se o *ArrayList* contém o elemento *x* especificado e false, caso contrário.
get(i)	Retorna o elemento no índice *i* especificado.
remove(i)	Remove o elemento no índice *i* especificado.
size( )	Retorna o número de elementos armazenados em *ArrayList*.

Exemplo 1 com a Classe *ArrayList*

```java
import java.util.ArrayList;

public class Principal{

  public static void main(String[] args){

    ArrayList<String> nomes = new ArrayList<String>();
    nomes.add("Luiz Henrique");
    nomes.add("Marco Aurélio");
    nomes.add("Paulo Roberto");

    System.out.println(nomes.get(1)); //Saída: Marco Aurélio
    System.out.println(nomes.contains("Paulo Roberto")); //Saída: true
  }
}
```

Precisamos importar a classe do pacote *java.util*.

Adicionando elementos no final do *ArrayList*.

Utilizando o método *get* para retornar o elemento na posição 1.

Utilizando o método *contains* para verificar se o elemento "Paulo Roberto" está no ArrayList.

Percorrendo um *ArrayList*

Podemos percorrer um *ArrayList* da mesma maneira que utilizamos para percorrer um vetor (*array*), ou seja, usando uma estrutura de repetição ***for***.

```
ArrayList<String> nomes = new ArrayList<>();
for(int i = 0; i < nomes.size(); i++){
    System.out.println(nomes.get(i));
}
```

No entanto, Java nos permite fazer a mesma coisa usando uma "nova" estrutura de repetição denominada ***for-each*** (para cada).

```
ArrayList<String> nomes = new ArrayList<>();
for(String item: nomes){
    System.out.println(item);
}
```

Em cada iteração, essa variável **item** referencia um elemento do ArrayList.

Vale observar que com o *for-each* não precisamos utilizar o índice para acessar cada posição do ArrayList.

Exemplo 2 com a Classe *ArrayList*

```java
import java.util.ArrayList;

public class Principal{

  public static void main(String[] args){

    ArrayList<String> nomes = new ArrayList<String>();
    nomes.add("Luiz Henrique");
    nomes.add("Marco Aurélio");
    nomes.add("Paulo Roberto");

    for(String e : nomes){
      System.out.println(e);
    }
  }
}
```

Utilizando o for-each para percorrer o *ArrayList*.

```
luiz@HP:~/TesteProgramas$ java Principal
Luiz Henrique
Marco Aurélio
Paulo Roberto
```

ArrayList de Tipos Primitivos

Vamos supor que eu queira usar um *ArrayList* para armazenar uma coleção de número inteiros.

- Para isso, coloco no meu programa a seguinte declaração:

  ArrayList<int> *num*;

- Isso vai funcionar?
  - **Não!**

```
Principal.java:7: error: unexpected type
    ArrayList<int> nomes = new ArrayList<int>();
              ^
  required: reference
  found:    int
```

  - A classe ArrayList aceita **somente tipos por referência** (objetos) como elementos.
  - O *int* em Java é um **tipo primitivo**.
- Como fazer então?

ArrayList de Tipos Primitivos

Como resolver o problema?

- O Java possui um mecanismo conhecido como *boxing*, que permite que **valores primitivos** sejam empacotados como **objetos** para uso com **classes genéricas**.
- Desse modo, para armazenar inteiros em um *ArrayList* usamos a classe *Integer*.

  ArrayList<Integer> *num* = **new ArrayList**<Integer>();
- E o compilador nos ajuda com o *autoboxing*, ou seja:
  - Sempre que necessário, o tipo *int* é **empacotado** como um objeto *Integer* ou um valor *int* dentro de um objeto *Integer* é **desempacotado**.
- Desse modo, declaramos um *ArrayList* de *Integer* mas podemos lidar com os dados normalmente como se fossem do tipo *int*.

ArrayList de Tipos Primitivos – Exemplo

```java
import java.util.ArrayList;

public class Principal{

  public static void main(String[] args){

    ArrayList<Integer> num = new ArrayList<Integer>();
    int x = 20;

    num.add(3);
    num.add(-5);
    num.add(x);

    for(int e: num){
      System.out.print(e + " ");
    }
    System.out.println();
  }
}
```

```
luiz@HP:~/TesteProgramas$ java Principal
3 -5 20
```

ArrayList de Tipos Primitivos

Cada tipo primitivo tem uma classe equivalente em Java:

- int $\longrightarrow$ Integer
- float $\longrightarrow$ Float
- double $\longrightarrow$ Double
- char $\longrightarrow$ Character
- boolean $\longrightarrow$ Boolean

ArrayList: Busca Usando uma "Chave"

Nós já vimos que podemos utilizar o índice para obtermos um elemento contido numa determinada posição do *ArrayList*.

```
ArrayList<Integer> notas = new ArrayList<>();
...
int x = notas.get(2);// Acessando o elemento na posição 2.
System.out.printf("A nota do aluno é: %d",x);
```

Mas e se eu desejar encontrar um elemento na lista a partir de alguma de suas características?

Por exemplo, em uma lista de contas bancárias eu preciso encontrar aquela cujo número é 1234.

Nesse caso, podemos usar uma estrutura de repetição (***for***) para percorrer o *ArrayList* procurando por uma "chave" de busca, ou seja, pelo valor desejado (no nosso exemplo desejamos encontar a conta bancária número 1234).

```
ArrayList<ContaBancaria> contas = new ArrayList<>();
…
public ContaBancaria buscarConta(int num){
    for(ContaBancaria c: contas){
        if(c.getNumero() == num) return c;
    }
    return null;
```

Retornando um objeto do tipo ContaBancaria

Procurando por uma conta com este número

Percorrendo a lista de contas bancárias

Se nenhuma conta da lista tiver o número "num", retornaremos o null para indicar que a conta não foi encontrada

ArrayList: Remoção de um Elemento

Nós também podemos utilizar o índice para remover um elemento contido numa determinada posição do *ArrayList*.

```
ArrayList<Integer> notas = new ArrayList<>();
notas.add(78);
notas.add(95);
notas.add(87);
notas.add(61);
System.out.println("Notas dos alunos:");
for(int n: notas){
    System.out.println(n);
}

notas.remove(1);//Removendo o elemento da posição 1.

System.out.println("Notas dos alunos após remoção:");
for(int n: notas){
    System.out.println(n);
}
```

Veja o resultado obtido ao executarmos esse código!

```
Notas dos alunos:
78
95
87
61
Notas dos alunos após remoção:
78
87
61
```

ArrayList: Remoção de um Elemento

Mas e se eu desejasse remover um elemento na lista a partir de alguma de suas características?

Por exemplo, em uma lista de contas bancárias eu quero remover aquela cujo nome do titular é *Reginaldo*.

Nesse caso, podemos usar uma estrutura de repetição (***for***) para percorrer o *ArrayList* procurando por uma "chave" de busca, ou seja, pelo valor desejado.

```
ArrayList<ContaBancaria> contas = new ArrayList<>();
…
public boolean removerConta(String nome){
    for(ContaBancaria c: contas){
        if(c.getNome().equals(nome)){
            contas.remove(c);
            return true;
        }
    }
    return false;
}
```

Retorno para indicar se a remoção ocorreu com sucesso ou não.

Percorrendo a lista de contas bancárias

Se uma conta da lista tiver sido removida

Se nenhuma conta da lista tiver sido removida

ArrayList: Ordenação dos Elementos

Para ordenar os elementos de uma coleção nós utilizaremos um método que está disponível na biblioteca de classes do Java.

Veja no exemplo a seguir a utilização do método de ordenação *sort*( ) implementado na classe *ArrayList*.

```
import java.util.ArrayList;
public class Principal{
    public static void main(String[] args){
        ArrayList<Integer> notas = new ArrayList<>();
        notas.add(78);
        notas.add(95);
        notas.add(87);
        notas.add(61);
        System.out.println("Notas dos alunos:");
        for(int n: notas){
            System.out.println(n);
        }

        notas.sort(null);

        System.out.println("Notas dos alunos após ordenação:");
        for(int n: notas){
            System.out.println(n);
        }
    }
}
```

```
Notas dos alunos:
78
95
87
61
Notas dos alunos após ordenação:
61
78
87
95
```

O parâmetro do método sort é um Comparator, o qual é usado para indicar como os elementos serão comparados entre si. No caso de tipos primitivos e strings não precisamos informar um Comparator (por isso usamos null).

ArrayList: Ordenação dos Elementos

Vamos aplicar a mesma ideia do exemplo anterior para ordenar uma coleção de contas bancárias?

```
import java.util.ArrayList;
public class Principal{
    public static void main(String[] args){
        ArrayList<ContaBancaria> contas = new ArrayList<>();
        contas.add(new ContaBancaria(5678,"Luiz"));
        contas.add(new ContaBancaria(9012,"Marco"));
        contas.add(new ContaBancaria(1234,"Paulo"));

        contas.sort(null);

        System.out.println("Nomes dos  após ordenação:");
        for(ContaBancaria c: contas){
            System.out.println(c.getNome());
        }
    }
}
```

Onde está o problema deste código?

Mas quando executamos esse código...

```
Exception in thread "main" java.lang.ClassCastException:
class ContaBancaria cannot be cast to class
java.lang.Comparable (ContaBancaria is in unnamed
module of loader 'app'; java.lang.
Comparable is in module java.base of loader 'bootstrap')
```

ArrayList: Ordenação dos Elementos

O problema no código anterior está no fato de que o Java não sabe qual é o critério de ordenação para objetos *ContaBancaria*.

Observe que diversos critérios podem ser usados para ordenar as contas bancárias, como por exemplo: número da conta, nome do titular etc.

Portanto, precisamos passar para o método *sort* um comparador (Comparator) que definará como os elementos dessa lista serão comparados.

Veja a seguir como seria a criação do objeto Comparator para ordenarmos as conta bancárias de acordo com o nome do titular das mesmas.

```
import java.util.Comparator;
...
Comparator<ContaBancaria> comparador = new Comparator<>( ){
   @Override
   public int compare(ContaBancaria cb1, ContaBancaria cb2){
      return cb1.getNome( ).compareTo(cb2.getNome( ));
   }
};
...
contas.sort(comparador);
```

Não esqueça de importar a classe Comparator

Como queremos comparar objetos da classe ContaBancaria, isso deve ser indicado aqui

Crie um objeto Comparator para indicar como os elementos da lista serão comparados entre si

Utilizamos este método para definir o critério de comparação

Aqui definimos o critério a ser usado na comparação, que no caso deste exemplo será o nome do titular da conta

Agora basta passar o objeto comparador criado como parâmetro para o método sort

ArrayList: Ordenação dos Elementos

Veja como fica o código completo da ordenação da lista de contas bancárias utilizando o nome do titular da conta como critério de ordenação.

```
import java.util.ArrayList;
import java.util.Comparator;
public class Principal{
   public static void main(String[ ] args){
      ArrayList<ContaBancaria> contas = new ArrayList<>( );
      contas.add(new ContaBancaria(5678,"Marco"));
      contas.add(new ContaBancaria(1234,"Paulo"));
      contas.add(new ContaBancaria(9012,"Luiz"));

      Comparator<ContaBancaria> comparador = new Comparator<>( ){
         @Override
         public int compare(ContaBancaria cb1, ContaBancaria cb2){
            return cb1.getNome( ).compareTo(cb2.getNome( ));
         }
      };
      contas.sort(comparador);
      System.out.println("Nomes dos titulares após ordenação:");
      for(ContaBancaria c: contas){
         System.out.println(c.getNome( ));
      }
   }
}
```

```
Nomes dos titulares após ordenação:
Luiz
Marco
Paulo
```

ArrayList: Ordenação dos Elementos

Agora vamos alterar o critério de ordenação das contas bancárias para o número da conta. Como ficará o código?

Como o número da conta é um *int* (não é uma classe), ele não possui o método *compareTo*. Nesse caso, teremos que adotar a estratégia apresentada a seguir.

```
import java.util.Comparator;
...
Comparator<ContaBancaria> comparador = new Comparator<>( ){
   @Override
   public int compare(ContaBancaria cb1, ContaBancaria cb2){
      return cb1.getNumero( ) - cb2.getNumero( );
   }
};
...
contas.sort(comparador);
```

Observe que o método ***compare*** implementado acima sempre retorna um inteiro. Isso porque ele funciona da seguinte forma:

- O método deve retornar um valor **positivo** quando o primeiro objeto for **maior do que** o segundo.
- O método deve retornar um valor **negativo** quando o primeiro objeto for **menor do que** o segundo.
- O método deve retornar o valor **zero** quando os objetos forem **iguais**.

Encapsulamento de Coleções

Analise o exemplo a seguir, onde a classe **Agencia** possui uma coleção de **contas bancárias**. Você enxerga algum problema nessa implementação?

```java
import java.util.ArrayList;

public class Agencia{
    private int codigo;
    private ArrayList<ContaBancaria> contas;

    public Agencia(int codigo){
        this.codigo = codigo;
        contas = new ArrayList<>();
    }

    public void adicionarConta(ContaBancaria cb){
        if(!contaCadastrada(cb.getNumero())){
            contas.add(cb);
        }
    }

    public boolean contaCadastrada(int num){
        for(ContaBancaria c: contas){
            if(c.getNumero() == num) return true;
        }
        return false;
    }

    public ArrayList<ContaBancaria> getContas(){
        return contas;
    }
}
```

```java
public class ContaBancaria{
  private int numero;
  private String nome;
  private float saldo;

  public ContaBancaria(int umNumero, String umNome){
    numero = umNumero;
    nome = umNome;
  }

  public int getNumero(){
    return numero;
  }

  public void depositar(float valor){
    saldo = saldo + valor;
  }

  public void sacar(float valor){
    saldo = saldo - valor;
  }
}
```

Encapsulamento de Coleções

A classe **Principal** testa a implementação das classes **Agencia** e **ContaBancaria**. Veja o resultado obtido com a execução desse código.

```
public class Principal{
    public static void main(String[] args){
        Agencia ag = new Agencia(0364);

        ContaBancaria conta1 = new ContaBancaria(5678,"Marco");
        ContaBancaria conta2 = new ContaBancaria(1234,"Paulo");
        ContaBancaria conta3 = new ContaBancaria(9012,"Luiz");

        ag.adicionarConta(conta1);
        ag.adicionarConta(conta2);
        ag.adicionarConta(conta3);

        System.out.println("Contas cadastradas:");
        for(ContaBancaria c: ag.getContas()){
            System.out.println(c.getNumero());
        }
    }
}
```

```
Contas cadastradas:
5678
1234
9012
```

Encapsulamento de Coleções

E se alterarmos a classe **Principal** da maneira apresentada a seguir? Qual será o resultado obtido com a execução desse código?

```
import java.util.ArrayList;
public class Principal{
    public static void main(String[] args){
        Agencia ag = new Agencia(0364);

        ContaBancaria conta1 = new ContaBancaria(5678,"Marco");
        ContaBancaria conta2 = new ContaBancaria(1234,"Paulo");
        ContaBancaria conta3 = new ContaBancaria(9012,"Luiz");

        ag.adicionarConta(conta1);
        ag.adicionarConta(conta2);
        ag.adicionarConta(conta3);

        ArrayList<ContaBancaria> contas = ag.getContas();
        contas.add(conta1);

        System.out.println("Contas cadastradas:");
        for(ContaBancaria c: ag.getContas()){
            System.out.println(c.getNumero());
        }
    }
}
```

Uma conta bancária foi cadastrada duas vezes?!

```
Contas cadastradas:
5678
1234
9012
5678
```

Encapsulamento de Coleções

E se alterarmos a classe **Principal** da maneira apresentada a seguir? Qual será o resultado obtido com a execução desse código?

Como resolver este problema?

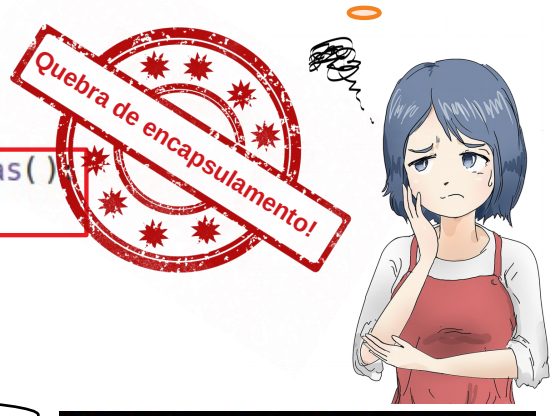
```
import java.util.ArrayList;
public class Principal{
    public static void main(String[] args){
        Agencia ag = new Agencia(0364);

        ContaBancaria conta1 = new ContaBancaria(5678,"Marco");
        ContaBancaria conta2 = new ContaBancaria(1234,"Paulo");
        ContaBancaria conta3 = new ContaBancaria(9012,"Luiz");

        ag.adicionarConta(conta1);
        ag.adicionarConta(conta2);
        ag.adicionarConta(conta3);

        ArrayList<ContaBancaria> contas = ag.getContas();
        contas.add(conta1);

        System.out.println("Contas cadastradas:");
        for(ContaBancaria c: ag.getContas()){
            System.out.println(c.getNumero());
        }
    }
}
```



Uma conta bancária foi cadastrada duas vezes?!

```
Contas cadastradas:
5678
1234
9012
5678
```

Encapsulamento de Coleções

Se você realmente precisa de um método que retorne a lista de contas bancárias, a estratégia é fornecer uma versão "somente-leitura" da lista!

E você pode fazer isso utilizando o método ***unmodifiableList***, que é um método estático da classe ***Collections***.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
public class Agencia{
    private int codigo;
    private ArrayList<ContaBancaria> contas;

    public Agencia(int codigo){
        this.codigo = codigo;
        contas = new ArrayList<>();
    }

    public void adicionarConta(ContaBancaria cb){
        if(!contaCadastrada(cb.getNumero())){
            contas.add(cb);
        }
    }

    public boolean contaCadastrada(int num){
        for(ContaBancaria c: contas){
            if(c.getNumero() == num) return true;
        }
        return false;
    }

    public List<ContaBancaria> getContas(){
        return Collections.unmodifiableList(contas);
    }
}
```

O método unmodifiableList retorna uma lista (do tipo List).

Perguntas?